TOUCH: In-Memory Spatial Join by Hierarchical Data-Oriented Partitioning

Sadegh Nobari[¶†] Farhan Tauheed[†‡] Thomas Heinis[†]
Panagiotis Karras[§] Stéphane Bressan[¶] Anastasia Ailamaki[†]
[¶]National University of Singapore, Singapore
[†]Data-Intensive Applications and Systems Lab, École Polytechnique Fédérale de Lausanne, Switzerland
[‡]Brain Mind Institute, École Polytechnique Fédérale de Lausanne, Switzerland
[§]Department of Management Science and Information Systems, Rutgers University, USA

ABSTRACT

Efficient spatial joins are pivotal for many applications and particularly important for geographical information systems or for the simulation sciences where scientists work with spatial models. Past research has primarily focused on disk-based spatial joins; efficient in-memory approaches, however, are important for two reasons: a) main memory has grown so large that many datasets fit in it and b) the in-memory join is a very time-consuming part of all disk-based spatial joins.

In this paper we develop **TOUCH**, a novel in-memory spatial join algorithm that uses hierarchical data-oriented space partitioning, thereby keeping both its memory footprint and the number of comparisons low. Our results show that TOUCH outperforms known in-memory spatial-join algorithms as well as in-memory implementations of disk-based join approaches. In particular, it has a one order of magnitude advantage over the memory-demanding state of the art in terms of number of comparisons (i.e., pairwise object comparisons), as well as execution time, while it is two orders of magnitude faster when compared to approaches with a similar memory footprint. Furthermore, TOUCH is more scalable than competing approaches as data density grows.

Categories and Subject Descriptors

H.2.8 [**DATABASE MANAGEMENT**]: Spatial databases and GIS; G.2.2 [**DISCRETE MATHEMATICS**]: Graph Theory

Keywords

Scalable algorithms; Spatial joins; TOUCH; Indexing

1. INTRODUCTION

Many applications dealing with spatial data rely on the efficient execution of spatial or distance joins. In geographical applications these joins are used to detect collisions or proximity between geographical features [30], i.e., landmarks, houses, roads, etc. and in medical imaging, spatial joins are used to determine which cancerous cells are within a certain distance of each other [10]. In the simulation sciences, where scientists build and simulate precise spatial models of the phenomena they are studying, distance joins are, for example, used to monitor the folding process of peptides [12].

Unfortunately, a distance join on two unsorted and unindexed datasets is a computationally costly operation, even if executed in the main memory of a supercomputer. Data models grow fast and, as a result, the distance join is a bottleneck in many scientific applications today, preventing them from scaling to bigger models. Apart from growing, data models in real-world scientific applications also become increasingly realistic and hence denser. The growing density of the models substantially increases the join's selectivity, rendering the efficient execution of this operation key for scaling to larger and more realistic models.

To formulate the problem, we translate the distance join into a spatial join that tests pairs of objects for intersection. Formally, the distance join takes a distance ϵ and two spatial datasets A and B and finds all pairs of spatial objects $a \in A$ and $b \in B$ such that the distance between a and b is less than or equal to ϵ. To translate the problem into a spatial join, we increase the size of all objects in one dataset by ϵ and then test both datasets for intersecting objects [15].

Existing research on spatial joins has mostly focused on disk-based approaches. Spatial join techniques designed for use in memory hence lack efficiency and scalability. In this paper we develop TOUCH, a two-way spatial join approach that works efficiently in memory. TOUCH combines concepts from previous work and avoids their problems, i.e., excessive memory footprint and excessive number of comparisons. In particular, it uses a hierarchy to mitigate replication of elements and data-oriented partitioning to avoid excessive pairwise comparisons. Additionally, data-oriented partitioning further reduces the number of comparisons by not considering the objects spatially far from other objects.

Because of the limited work on in-memory spatial joins, we also draw inspiration from on-disk approaches and compare our approach to disk-based approaches used in memory. The latter is reasonable as growing memory capacities allow for approaches with a bigger memory footprint, originally designed for use on disk, to be used in main memory.

We apply our solution on the ”touch detection” problem, a challenging neuroscience application that arises in collaboration with computational neuroscientists. The neuroscientists build biophysically realistic models with data acquired during anatomical research of the rat brain. In their models, as in the rat brain, each neuron has branches extending into large parts of the tissue. The neurons receive and send information to other neurons using these branches. To determine where in the model to place the synapses (structure that permits a neuron to pass a signal to another neuron), it suffices to find the places where the distance between two branches is below a given threshold [18], i.e., where the neurons touch.

Our experiments show that TOUCH outperforms existing spatial joins algorithms in terms of number of comparisons and execution time. When compared to the fastest related approach TOUCH requires substantially less memory. In the context of our working example, TOUCH performs the spatial join at least one order of magnitude faster than related work. Our experiments also indicate that TOUCH will scale better to more detailed and denser neuroscience datasets in the future.

Our work has significant potential impact beyond neuroscience applications. Spatial joins are broadly used in many applications beyond neuroscience and the simulation sciences, and their use in memory is becoming increasingly important in many applications for two reasons. First, main memory has grown so large that many datasets fit into it directly and the spatial join can be entirely performed in memory. Second, the in-memory join is also an integral part of all disk-based joins. Disk-based joins partition datasets that do not fit in memory and then join the partitions in memory. Speeding up the in-memory join helps to considerably speed up on-disk approaches as a whole as well, particularly if the join is very selective and many objects need to be compared (thus spending a large share of the overall time for the in-memory join).

The remainder of the paper is structured as follows. We discuss related work and its shortcomings in Section 2. In Section 3 we motivate our work and in Section 4 we present TOUCH, our algorithm, and discuss its implementation in Section 5. We compare TOUCH to related approaches in Section 6 and draw conclusions in Section 7.

2. RELATED WORK

While several spatial join approaches have been developed for disk in the past, only few have been developed for use in memory. Many disk-based spatial join algorithms, however, can also be used in memory. In the following we discuss all related work no matter if it has traditionally been used for disk or in memory.

2.1 In-Memory Approaches

Only two approaches have been developed for use in memory and thus have a small memory footprint: the nested loop join [24] and the plane-sweep join [28].

The nested loop join iterates over both spatial datasets in a nested loop and compares all pairs of objects. While this results in a complexity of $O(n^2)$, no additional data structures are needed, making the approach very space efficient.

The plane-sweep approach sorts the datasets in one dimension and scans both datasets synchronously. All objects on the sweep plane (stored in an efficient data structure) are compared to each other to see if they overlap. Because the objects are only sorted in one dimension, objects which are not near each other in the other dimensions may be on the sweep plane at the same time, thus leading to redundant comparisons and hence slowing down the approach.

Despite its deficiencies, however, the plane-sweep approach is still broadly used to join in memory the partitions resulting from disk-based spatial joins.

2.2 On-disk Approaches

Because they can also be used in memory, in the following we discuss distance join approaches designed for disk. We categorize the approaches based on whether they require an index on both, one, or none of the datasets.

2.2.1 *Both Datasets Indexed*

If both datasets A and B are indexed with R-Trees [13], a synchronous traversal [7] can be used to join them. With the R-Tree indexes I_A and I_B on datasets A and B, this approach starts from the roots of the trees and synchronously traverses the tree to the leaf level. If two nodes $n_A \in I_A$ and $n_B \in I_B$ on the same level (one from each tree) intersect, then the children of n_A will be tested for intersection with the children of n_B. This process recursively traverses the trees to the leaf level where the objects are compared.

By building on the R-Tree, this approach also inherits the problems of the R-Tree, namely inner node overlap and dead space. Overlap in the R-Tree structure leads to too many comparisons and hence slows down the join operation. Extensions like the R*-Tree [6] or the R+-Tree [29] have been proposed to reduce overlap. The former tackles overlap with an improved node split algorithm (reinsertion of spatial objects if a node overflows) while the latter duplicates objects to reduce overlap. Duplicating objects, however, also leads to duplicate results which have to be filtered.

Arguably the most efficient R-Trees can be built through bulkloading if the data is known a priori. Several bulokloading approaches like the STR [19], Hilbert [16], TGS [11] and the PR-Tree [4] have been developed, all yielding better performance than R+-Tree or R*-Tree. The R-Tree resulting from bulkloading with Hilbert and STR perform similarly and outperform TGS as well as the PR-Tree on real-world data. TGS and the PR-Tree, however, outperform STR and Hilbert on data sets with extreme skew and aspect ratio.

Double index traversals are also possible with Quadtrees [2] (or Octrees in 3D). Similar to the R+-Tree objects are duplicated (or references to the objects) and duplicate results are possible and need to be filtered at the end [3].

2.2.2 *One Dataset Indexed*

Extending on a basic nested loop join approach, the indexed nested loop join [9] requires an index I_A for dataset A. The approach loops over dataset B and queries I_A for every object $b \in B$. Executing a query for each object is a substantial overhead, particularly if $B >> A$.

The seeded tree approach [21] also requires one dataset to be indexed with an R-Tree. The existing R-Tree I_A on dataset A is used to bootstrap building the R-Tree I_B on dataset B. After building the second index I_B, a synchronous traversal [7] is used for the join. Using the structure of I_A to build I_B ensures that the bounding boxes of both indexes are aligned, thereby reducing the number of bounding boxes that need to be compared. Improvements avoid memory thrashing [23] or use sampling to speed up building the R-Tree [20].

2.2.3 *Unindexed*

Disk-based approaches first partition both datasets and then join the resulting partitions in-memory. When assigning spatial objects to partitions, some objects may intersect with several partitions. Two different approaches, *multiple assignment* and *multiple matching*, have been developed to deal with this ambiguity.

Multiple Assignment: this strategy assigns each spatial object to all partitions it overlaps with (through duplication). The advantage is that the distance join only needs to compare objects inside one partition with each other (and not across partitions). Duplication, however, has major drawbacks, namely a) more comparisons need to be performed and b) because result pairs may be detected twice they need to be deduplicated either at the end (by keeping all results, thereby increasing the memory used - and deduplicating them at the end) or throughout the join [8].

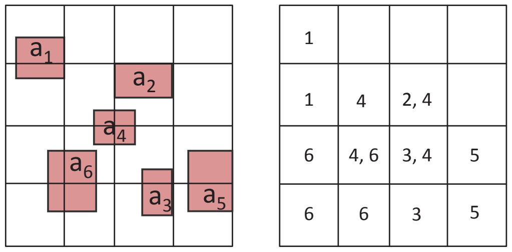


Figure 1: PBSM partitioning (left) and assignment (right)

PBSM [27] is the most recent and comparatively most efficient multiple assignment approach. As shown in Figure 1, PBSM partitions the entire space of both datasets into cells using a uniform grid. Every object from dataset A is assigned to all cells c_A it overlaps and all $b \in B$ are assigned to cells c_B respectively. After assigning all objects to cells, all pairs of cells c_A and c_B which have the same position are compared, i.e., all objects assigned to c_A and c_B are tested for intersection. Because objects are replicated intersections may be detected multiple times and hence deduplication needs to be performed.

The non-blocking parallel spatial join NBPS [22] algorithm produces the result tuples continuously as they are generated. NBPS distributes the tuples of the join relations to the data server nodes according to a spatial partitioning function. In contrast to PBSM, NBPS avoids duplicates so that the result can be returned immediately. More precisely, NBPS uses a revised reference point method to avoid the duplicates while TOUCH uses a hierarchical partitioning that not only avoids any replication (in comparison with NBPS) but also avoids the duplicates from the first stage of the distribution.

Multiple Matching: this strategy on the other hand assigns each spatial object only to one of the partitions it overlaps with. Hence when joining, objects in several partitions must be compared with each other, as an object at the border of one partition can potentially intersect with an object at the border of an adjacent partition.

The Scalable Sweeping-Based Spatial Join [5] is similar to PBSM but avoids replication. It partitions space into n equi-width strips in one dimension and maintains for every strip two sets LA_n and LB_n. It assigns each $a \in A$ that entirely fits into strip n to LA_n and for B and LB_n respectively. Finally, it uses an in-memory plane-sweep to find all intersecting pairs from LA_n and LB_n for all n.

An object o intersecting several strips will not be replicated but will instead be assigned to sets LA_{jk} and LB_{jk} respectively where j is the strip where o starts and k where o ends. When joining LA_n and LB_n all sets LA_{jk} and LB_{jk} with $j \leq n \leq k$ will also be considered in the plane-sweep.

To avoid the replication of objects, S3 [17] maintains a hierarchy of L equi-width grids of increasing granularity as

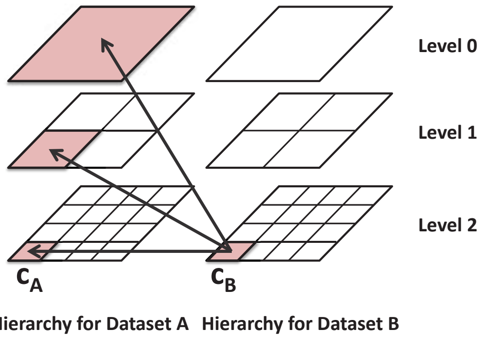


Figure 2: S3 space partitioning and multi-level join.

shown in Figure 2. In D dimensions the grid on a particular level l has $(2^l)^D$ grid cells and assigns each object of both datasets to a grid cell in the lowest level where it only overlaps one cell. To obtain this assignment, the algorithm starts with level L (Level 2 in figure 2) and moves up the levels until it finds the level where the object only overlaps one cell.

The algorithm maintains two hierarchies, H_A for dataset A and H_B for dataset B. Once all objects are assigned, the cells of H_A and H_B are joined. More precisely, a cell c_B of H_B is joined with its corresponding cell c_A of H_A and all the cells on higher levels of H_A enclosing c_A (example cells are shaded in Figure 2). Joining a cell with its counterpart as well as with cells on higher levels is repeated on all levels.

The process of joining the cells implies that the objects assigned to the highest level will be compared to all other objects (on all lower levels) and hence the more objects are assigned to the highest level, the more comparisons will be needed. Datasets assigning more objects to levels closer to the leaf level will require fewer comparisons for the join.

3. CHALLENGE

The development of TOUCH is driven by the requirements of the computational neuroscientists we collaborate with. In order to better understand how the brain works, the neuroscientists build biophysically realistic models of a neocortical column on the molecular level and simulate them on a BlueGene/P with 16K CPUs. Each of the models contains several thousand neurons where each neuron and its branches are modeled as thousands of cylinders. Figure 3 shows a cell morphology, with cylinders modeling the dendrite and axon branches in three dimensions.

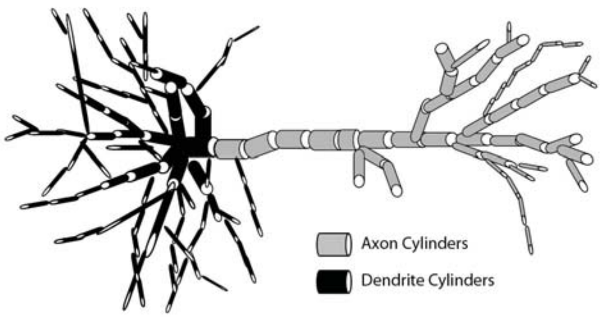


Figure 3: Neuron modeled with cylinders.

The models are built based on the analysis of real rat brain tissue. The structure of the neuron is rebuilt using brightfield microscopy and their electrophysiological properties are obtained by the patch clamp technique [14]. The locations of synapses (i.e., points where impulses leap over between neurons), however, are not known, cannot be determined by the above techniques and thus need to be added in a post-processing step. Placing the synapses is a one-off operation executed only once for each model built. The synapse locations are determined by the following rule: a synapse is placed wherever a neuron's dendrite is within a certain distance of another neuron's axon. Previous neuroscience research has confirmed that a realistic model of the brain is built by following this rule [18].

The problem of placing synapses therefore translates to a spatial distance join between two unindexed and unsorted datasets, one dataset containing cylinders representing axons and one containing cylinders representing dendrites. The distance join is only executed once on each new model and thus no data structures can be shared between distance joins on different models. Because the 16K cores of the BlueGene/P cannot access the disk concurrently, the join has to be performed in memory alone. Furthermore, this independency is advantageous for data-parallel algorithms that can be executed on GPUs [25]. To do so, and because this is an embarrassingly parallel problem, the dataset is split into 16K contiguous subsets, each subset is loaded in the memory of a core and the distance join is performed locally (independent of the other cores and thus massively parallel).

The models currently built and simulated by the neuroscientists contain at most few million neurons, but the ultimate goal is to simulate the human brain with approximately 10^{11} neurons. To achieve this goal, the number of neurons needs to increase until the model has the same neuron and synapse density as the human brain. The higher density of a dataset modeling the human brain, as opposed to those modeling the rat brain, will substantially increase the selectivity of the distance join between the two datasets, as more synapses are found in the former than in the latter. An efficient distance join approach is therefore pivotal already today and the advancement of neuroscience research requires an in-memory spatial join that will scale to the increasingly high selectivity of arising models of the brain/neuroscience datasets.

The particular application notwithstanding, an efficient in-memory spatial join is important for many applications: after all, every disk-based spatial join needs to perform an in-memory join.

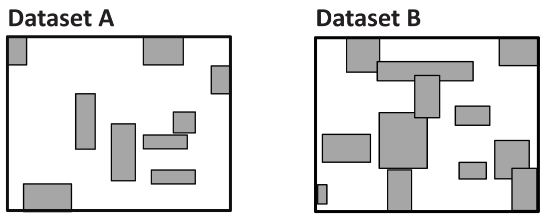
4. TOUCH

Given the lack of adequate in-memory spatial join approaches we develop TOUCH, a new in-memory spatial join algorithm for two unsorted and unindexed spatial datasets, DS_1 and DS_2. We translate the distance join required by our motivating example into a spatial join that detects intersection between objects: instead of finding all pairs of objects $d_1 \in DS_1$ and $d_2 \in DS_2$ such that $distance(d_1, d_2) \leq \epsilon$, we increase the size of all objects of one dataset, say DS_1, by ϵ and test for intersection with objects of DS_2 [15].

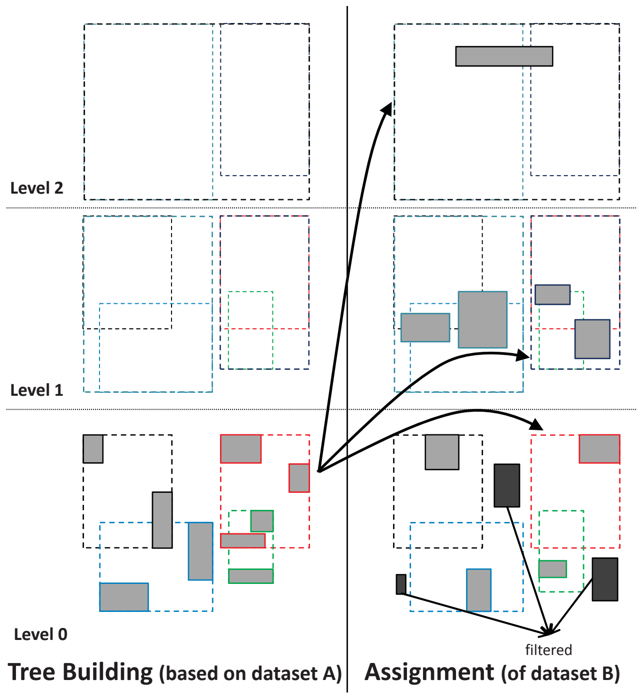
It is established practice to perform a spatial join in two phases: filtering followed by refinement [26]. Like most other approaches, TOUCH focuses on the filtering phase in which all objects are approximated by bounding boxes. Our solution can be combined with any off-the-shelf solution to the second refinement phase, which takes into account the exact object shapes (e.g., cylinders, spheres, etc.).

4.1 TOUCH Ideas

In developing TOUCH, we are inspired from previous disk-based approaches, which can also be used in main memory.



(a) The datasets A and B



(b) Tree building, assignment and joining phases

Figure 4: The three phases of TOUCH: building the tree, assignment and joining.

However, we aim to combine the benefits and avoid the pitfalls of previous approaches. In particular, we want to avoid multiple assignment (as in PBSM), because it replicates objects and therefore (a) increases the memory footprint; (b) requires multiple comparisons; and (c) makes it necessary to deduplicate the results.

At the same time we want to reduce the number of comparisons of multiple matching approaches and we therefore want to use data-oriented partitioning instead of space-oriented partitioning as S3 does.

To further reduce the number of comparisons, we also use filtering, a concept used by S3. In S3, objects from the second dataset B are discarded if they intersect only with cells that contain no objects from dataset A. If object $b \in B$ is only overlapping cells that contain no object from A then b cannot possibly intersect with any object from A. Hence b does not need to be considered further.

The main innovation of TOUCH lies in the fact that it directly assigns objects of the second data set to the data-oriented index of the first. By doing so it avoids the problems caused by excessive index-overlap in other data-oriented approaches (R-Tree) as well as the problems of space-oriented indexing approaches (like S3). As we will explain, the combination of data-oriented partitioning during index-building on the first dataset with hierarchical assignment of the second dataset leads to significantly fewer comparisons and speeds up the join itself.

The fact that we design our approach for use in memory gives us more degrees of freedom. We no longer have to align the data structures for the disk page size, but can choose the size of the data structures used more flexibly (partitions of arbitrary size, variable fanout, etc.).

4.2 Algorithm Overview

TOUCH is organized in three phases. During its first phase it builds a hierarchical tree as support data structure (like an R-Tree) and indexes dataset A with it. In particular, it stores the objects of A in the leaf nodes of the tree. In the second phase, all objects in dataset B are assigned to the inner nodes (non-leaf nodes) of the tree. The third phase uses a local join to join the datasets by comparing the objects $b \in B$ of each inner node n with the objects $a \in A$ in the leaf nodes reachable from n. Algorithm 1 gives an overview.

Algorithm 1: TOUCH

```
Input: $A, B$: two spatial datasets;
       $p$: number of partitions
Data: $P_A$: partitions built on $A$
Output: $R$: Result pairs
1 $R = \emptyset$;
2 Group the objects of $A$ into $p$ partitions $P_A$;
3 $T$ = Construct the hierarchical partitioning tree on $P_A$;
4 Assign the objects of $B$ to $T$;
5 foreach $in \in innernodes\ of\ T$ do
6     foreach $leaf \in descendent\ leaf\ nodes\ of\ in$ do
7         $R \leftarrow R \cup$ join($in.entities$ , $leaf.entities$);
8 Return $R$;
```

4.3 Tree Building Phase

In the first phase TOUCH builds a hierarchical tree based on dataset A. Each node of this tree contains pointers to *children* nodes (the number of children each node has is called the fanout), an *MBR* (minimum bounding rectangle) and *entities*, i.e., pointers to objects of *either* dataset A *or* dataset B. To build the tree, TOUCH groups spatially close objects of dataset A based on their MBRs into buckets of equal size. Different strategies can be used to assign spatially close objects to the same bucket, TOUCH uses STR [19].

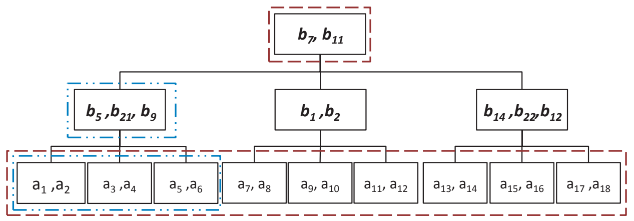


Figure 5: TOUCH's tree data structure.

Each bucket is used as a leaf node, i.e., a leaf node references as the entities the objects $a \in A$ in a bucket. The tree is then built recursively: starting from the leaf level l, on the next higher level $l-1$, f nodes of N_l from level l are summarized into a new node (f is the fanout). The new node has an MBR that encloses the MBRs of the f nodes from N_l. Building the tree finishes with the root node which has an MBR that encloses all objects of dataset A. An example of a resulting tree is illustrated in Figure 5. The dashed lines indicate that the MBR of each node encloses all the MBRs of its children. Algorithm 2 illustrates the tree building phase.

Algorithm 2: Tree Building Phase

```
Input: objs: spatial objects of dataset A
       fo: fanout of the tree
Output: tree: root of tree
Data: $P_A$: array of partitions of nodes/objects
      nodes: array of nodes
1 $P_A$ = partition $objs$ into partitions of size $fo$;
2 while $|P_A| > 1$ do
3     foreach $partition\ p \in P_A$ do
4         calculate $MBR$ of $p$;
          make new node $n$ with $MBR$
          set objects inside $p$ as children of $n$;
          insert $n$ into $nodes$ ;
5     $P_A$ = partition $nodes$ into partitions of size $fo$;
6 Return $P_A$;
```

Figure 4(b)(left) shows how the tree is built based on dataset A in Figure 4(a). Starting at level 0, several objects (up to the fanout, three in this example) are grouped into a node in the next higher level indicated by the dashed rectangle. Objects (or nodes on higher levels) are recursively grouped until the process arrives at the root node.

Should one of the datasets already be indexed with a hierarchical index which uses data-oriented partitioning, then this index can easily be converted to the tree needed for TOUCH and the tree building phase can be skipped.

4.4 Assignment Phase

After constructing the tree, in which the leaf nodes point to the objects of dataset A and each node has an MBR that is defined recursively based on the MBRs of dataset A, Algorithm 3 distributes the objects of dataset B to the nodes of the tree based on the nodes' MBRs.

The assignment is based on the number of MBRs on a level the object b overlaps with (we define overlap as both intersection and containment). In order to assign an object b of dataset B, the algorithm starts from the root node. At each level, object b can overlap with the MBR of none, one or several nodes:

- **Overlap with no MBR:** if b overlaps with none of the MBRs then we can safely *filter* b. Because b does not overlap with any of the nodes' MBR, it will not intersect with any of the objects in the node either and can hence be excluded from further consideration.
- **Overlap with one MBR:** if b overlaps with only a single node's MBR then the algorithm recursively checks if b overlaps with the children of the node in order to find the lowest level where b overlaps with a single node's MBR.
- **Overlap with several MBR:** if b overlaps with more than one of the nodes' MBR then the object will be assigned to the parent of these nodes.

In brief, the algorithm assigns an object $b \in B$ to the lowest (closest to the leaf nodes) inner node n in the tree whose MBR overlaps b and has no siblings overlapping it. Algorithm 3 illustrates how the objects of dataset B are assigned to the nodes in the tree built during the first phase.

Figure 4(b)(right) shows the assignment phase where objects are tried to be assigned to level 1. Objects are recursively assigned one level higher (closer to the root) as long as they overlap with several nodes (in this example only one object needs to be assigned to the root node). If objects are outside the MBR of all nodes on the leaf level, they can safely be filtered (the black objects on level 0) because they cannot intersect with any object from dataset A.

Algorithm 3: Assignment Phase

```
Input: $obj$: spatial object;
       $T$: partitioning tree
Output: $p$: the inner node of $T$ to assign $obj$ to
$overlap = false$;
$p$ = root node of $T$;
while $p$ is not a leaf node do
    foreach $ch \in$ children of $p$ do
        $overlap = false$;
        if $obj.MBR$ overlaps with $ch.MBR$ then
            if $overlap$ then
                Return parent of $p$;
            else
                $p = ch$;
                $overlap = true$;
    if $\neg overlap$ then
        Return $\phi$; // $obj$ is filtered.
Return $p$;
```

4.5 Join Phase

With all objects from dataset A referenced by the leaf nodes and all objects of dataset B by the inner nodes, the actual join is performed. Each inner node n is joined with all its successors, i.e., with all nodes that have an MBR enclosed by the MBR of n. The example in Figure 4(b) shows with what nodes on the right, an example node on the left needs to be compared (illustrated with the curved arrows).

As for S3, if all objects of B are assigned to the root node then all objects of dataset A are compared against all objects of B. However, in an ideally distributed dataset, objects of B will be assigned as close to the leafs as possible; the more objects are assigned to nodes closer to the leaf nodes, the fewer objects ultimately need to be compared to each other.

We join objects of the inner nodes with the objects in their descendant leaf nodes using Algorithm 4. We partition the space into an equi-width grid. Then we assign the objects of the leaf nodes to all cells they overlap. Ultimately, we compare each object of an inner node with only the objects in the grid cells it overlaps with.

Algorithm 4: Join Phase

```
Input: $inner$: an inner node
Output: $R$: a subset of result pairs
Uniformly divide the space to cells of same size;
foreach $obj \in inner$ do
    $C$ = cells that $obj$ overlaps with;
    foreach $cell \in C$ do
        $cell.objects \leftarrow cell.objects \cup obj$;
foreach $L \in$ descendant leaf nodes of inner do
    foreach $obj_a \in L$ do
        $C$ = cells that $obj_a$ overlaps with;
        foreach $cell \in C$ do
            foreach $obj_b \in cell.objects$ do
                if $obj_a$ overlaps $obj_b$ then
                    $R \leftarrow R \cup (obj_a, obj_b)$;
Return $R$;
```

The result of the join performed in this last phase of TOUCH is exactly the set of pairs of objects o_1 and o_2 where the MBRs of o_1, o_2 overlap. TOUCH does not produce any more candidate pairs than other approaches do.

4.6 Proof of Correctness

The correctness of TOUCH is proven in Theorem 1. Lemma 3 proves that the result pairs generated by TOUCH (until joining the buckets, i.e. the global join) are unique. Without loss of generality, in the following we assume that a pair of objects (a, b), $a \in A$ and $b \in B$, is in the result pairs iff MBR of object a overlaps the MBR of object b. In the case of a distance join with a distance of ϵ, we can extend the objects of a dataset by ϵ then check the overlapping of the objects in the join.

LEMMA 1 (COMPLETENESS). *The assignment phase of TOUCH does not miss any pair of overlapping objects.*

PROOF. Assume that object a of dataset A overlaps object b of dataset B and the pair (a, b) is not in the result set of TOUCH. Because in the probing phase of TOUCH, a is compared against the bucket it overlaps at each level, and b overlaps with a, then b must be inside at least one of these buckets. As a result, b must be compared against a when a's bucket is joined against the bucket that b is assigned to. □

LEMMA 2 (SOUNDNESS). *The set of result pairs generated by TOUCH is a subset of pairs of overlapping objects.*

PROOF. When a pair of objects (a, b), $a \in A$ and $b \in B$, is not overlapping, then two scenarios are possible. First, the buckets a and b are assigned to are not overlapping, hence the buckets are not joined with each other. The other possible scenario is that the buckets a and b are assigned to are overlapping. As a result, the objects inside the buckets are joined with each other. However, because (a, b) are not overlapping, this pair of objects is not reported as result. Any pair in the set of result pairs of TOUCH hence overlap. □

THEOREM 1. *TOUCH finds all object pairs that overlap.*

PROOF. The proof follows from Lemmas 1 and 2. □

LEMMA 3 (NO DUPLICATION). *The set of result pairs generated by TOUCH (until joining the buckets, i.e. the global join) contains unique pairs of objects.*

PROOF. Because each object b of the dataset B is assigned to at most one bucket (single assignment) closest to the leaf level that covers the object, b is compared against each object of dataset A at most once. Hence, for any b, any result pair that contains b appears only once. □

5. IMPLEMENTATION

In order to implement TOUCH we build in a first phase a tree based on dataset A. Each node in the tree contains a triple {*children*, *MBR*, *entities*}. The leaf nodes contain as entities the objects of dataset A and each node's MBR encloses all objects in it. The MBR of an inner node is computed recursively based on the MBRs of its children.

In the second phase, the objects $b \in B$ are distributed and stored as the entities of the inner nodes of the tree. Following Algorithm 3, each object b is assigned to the node with the smallest MBR that still covers it.

5.1 Partitioning

In the first phase TOUCH partitions the objects of dataset A into buckets and builds a tree, similar to an R-Tree [13].

In TOUCH we use the Sort-Tile-Recursive approach [19] to group the objects. This technique first sorts the MBRs on the first dimension. Then STR divides the dataset into $P^{1/D}$ slabs, where P is the number of partitions and D is the number of dimensions, according to their current order. Finally, STR recursively processes the remaining slabs using the remaining $D-1$ dimensions. STR typically produces leaf nodes with the smallest MBRs [19] (also on higher levels) and thus allows for more effective filtering.

5.2 Design Parameters

In the course of building the data structures, i.e., the tree, and ultimately joining the datasets several parameters can be set and tuned. In the following we discuss what they are and how they can have an impact on the performance.

5.2.1 *Tree Parameters*

The tree built on dataset A in the first phase of the join resembles an R-Tree and hence has tunable parameters like the size of the leaf nodes and the fanout, i.e., the number of children each node has (except the leaf nodes). The size of the inner nodes depends on the assignment of dataset B and can hence not be known a priori.

For the disk-based R-Tree, the fanout defines the node size and is chosen such that the nodes fit on an integer number of disk pages. Because TOUCH is developed for memory, disk restrictions no longer apply and we can choose fanout and node size more freely. The choice of the fanout, however, has an impact on the performance.

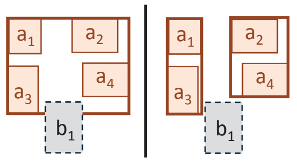


Figure 6: Filtering.

The smaller the fanout, the higher the tree and thus the better can the objects be distributed over the levels. Distributing them over more levels leads to fewer objects of the dataset B assigned to each node and therefore leads to fewer comparisons. Using the same approach on S3 would not be beneficial; it would lead to very fine-grained grids and hence many objects would be propagated to higher levels, leading to more comparisons.

A big fanout, say the maximum fanout when the root node connects with all leaf nodes, on the other hand degrades the performance of TOUCH. In this case the entire dataset B is assigned to the root node and all objects of A are compared against it, resulting in the maximum number of comparisons.

The fanout also has an impact on filtering. Objects of dataset B are filtered (and are thus not considered for the result set) if they do not intersect with any of the leaf nodes. As Figure 6 illustrates, the smaller the fanout, the fewer objects are in the leaf nodes, the smaller their MBRs will be (less dead space in the MBR that is actually not covered by any of the objects) and hence the more objects can be filtered (e.g., b_1 in Figure 6).

5.2.2 *Local Join Parameters*

The local join has the size of grid cells as a parameter. All objects assigned to one of the cells must be compared pairwise. Setting the cell size too big therefore means that more objects are assigned to each cell and have to be compared pairwise. Setting it too small on the other hand means too many objects need to be replicated.

To determine the grid cell size, we ensure that the cell size is considerably larger than the average size of the objects.

5.2.3 *Join Order*

Given two datasets A and B, deciding which dataset to use first to build the tree and which one to assign to the inner nodes of the tree is pivotal for performance.

To improve filtering, the decision would ideally be based on the density of both datasets: the sparser the first dataset, the more objects of the second dataset may be filtered. Not knowing the density of the datasets a priori, we can still make a reasonable assumption and take the smaller dataset as the first dataset. If it has the same (or a bigger) spatial extent than the bigger dataset, then it will be sparser and allow for more filtering. If its spatial extent is smaller than the one of the bigger dataset, then the difference in extent allows for effective filtering as well.

Using the smaller dataset first will also help to speed up building the hierarchical structure as well as reduce its size in memory.

6. EXPERIMENTAL EVALUATION

In this section we measure the performance of TOUCH and study the impact of dataset characteristics on its performance. We compare TOUCH against the two spatial joins designed for memory, plane-sweep and nested loop. Although the nested loop join is the textbook worst case of a join, we still include it in the comparison because it is broadly used (as part of disk-based joins and otherwise).

Because growing memory capacities allow for approaches with a bigger memory footprint, originally designed for use on disk, to be used in main memory we also compare TOUCH against approaches designed for disk but used in memory, namely S3 [17], PBSM [27], the indexed nested loop [9] and the synchronous R-Tree traversal [7] (RTree for short). For RTree we use an in-memory implementation of the bulk-loading STR R-Tree. The STR R-Tree exhibits the best performance for non-extreme (with respect to aspect ratio and skew) real world data [4].

To make the experiments reproducible we do not use the BlueGene/P, but execute them single-threaded on one core of a Linux machine. This is similar to TOUCH's use in the context of the neuroscience application: because the problem is embarrassingly parallel, each core of the BlueGene/P runs a spatial join on a subset of the entire dataset in its local memory, independent of the other cores.

6.1 Setup

Hardware: The experiments are run on a Linux Ubuntu v2.6 machine equipped with 2 quad CPUs AMD Opteron (each with 2 MB L2, i.e., 512KB $\times$ 4 cores and 6 MB L3 cache), 64-bit @ 2.7GHz and 64GB RAM. The storage consists of 4 SAS disks of 300GB capacity each, striped to 1TB.
Software: For all the experiments the OS can use the remaining memory to buffer disk pages. For a fair comparison the implementations of all approaches are single threaded. All approaches are implemented in C++.
Settings: We choose to join, in line with the discussion in Section 5.2.3, the smaller datasets first. In absence of heuristics to set the parameters of all competing spatial join approaches we determined the parameters with parameter sweeps. For the R-Tree based approaches (indexed nested loop and the synchronous R-Tree traversal) we use the best configuration, i.e., a fanout of 2 and nodes of 2KB and S3 is configured with a fanout of 3 and 5 levels. For PBSM we use two different configurations, illustrating the trade-offs involved: (1) PBSM-500, the fastest configuration according to the parameter sweep with a resolution of 500 cells in each dimension which, however, has a big memory footprint due to replication and (2) PBSM-100 a configuration with a resolution of 100 cells in each dimension that has a smaller memory footprint but takes longer to execute. For TOUCH we set the parameters in line with the discussion in Section 5.2, i.e., we set the fanout to 2, the number of partitions to 1024, and for the local join the number of cells in each dimension to 500. We empirically observed the best

Datasets	Size (objects)	$\epsilon = 5$	$\epsilon = 10$
Uniform	$160K \times 1600K$	73.4	87.1
Gaussian	$160K \times 1600K$	300	617.5
Clustered	$160K \times 1600K$	89.5	125.2
Neuroscience	$644K \times 1285K$	411	728

Table 1: Selectivity of the datasets ($\times E^{-6}$)

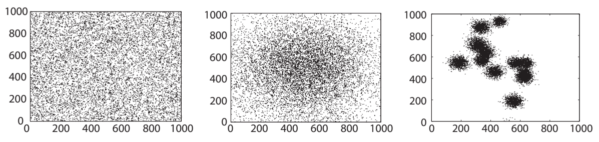
performance with these parameters. The time to build the indexing structures is included as part of the reported query execution times.

6.2 Experimental Methodology

For the experiments we use two fundamentally different types of datasets. To demonstrate the general applicability of our approach and to stress that we do not exploit any particularity of the neuroscience datasets, we primarily use 3D synthetic spatial datasets to evaluate TOUCH.

To generate synthetic 3D datasets we distribute spatial boxes with each side of uniform random length (between 0 and 1) in a constant space of 1000 space units in each of the three dimensions. We use three different distributions, namely *uniform*, *Gaussian* ($\mu = 500$, $\sigma = 250$) and *clustered* to generate datasets containing from 160K to 960K objects and from 1.6 to 9.6 million objects. The clustered distribution uniformly randomly chooses up to 100 locations in 3D space around which the objects are distributed with a Gaussian distribution ($\mu = 0$, $\sigma = 220$). In all experiments we always join datasets of the same type only. Projections of the three 3D distributions into 2D are shown in Figure 7.

Driven by our motivating example from neuroscience, we use two different ϵ, 5 and 10. Both these distance predicates are used to build models of the brain.



(a) Uniform (b) Gaussian (c) Clustered

Figure 7: Distributions used for the experiments.

To further characterize the datasets, Table 1 reports the selectivity [1] of both synthetic and neuroscience datasets according to equation 1.

$$Selectivity = \frac{|resultpairs|}{|datasetA| \times |datasetB|} \quad (1)$$

In addition to the synthetic datasets we measure the performance of the algorithms on a real dataset as well. We use a model of a small part of the rat brain represented with 450 million cylinders as objects. We take from this model a contiguous subset with a volume of $285\mu m^3$ and approximate the cylinders with minimum bounding boxes. For dataset A we use the 644,000 axon cylinders in the subset and for dataset B the 1.285 million dendrite cylinders in the subset (a realistic ratio between axon & dendrite cylinders). Synapses are placed where axon cylinders are in proximity of dendrite cylinders.

Like the experimental evaluations of PBSM [27], S3 [17], R-Tree-based join [7] and others we focus on the filtering phase of the join (where the objects are approximated by minimum bounding rectangles and are tested for intersection) and do not include the time for the refinement phase as this depends on the geometry of the objects. We measure for all approaches the total execution time as well as the implementation-independent number of comparisons, i.e., comparisons between the bouding boxes of two objects where they are tested for intersection.

Because the spatial join is developed for use in memory, we also measure the memory footprint. We compare TOUCH against the nested loop join (NL), plane-sweep join (PS), PBSM, S3, synchronous R-Tree traversal (RTree for short) and the index nested loop join, each of the latter four with the plane-sweep as the local join. Our implementation of PBSM deduplicates during the join [8] (and not at the end) and thus does not need additional memory.

Unless otherwise noted, the experimental methodology is similar to the one of PBSM [27] and S3 [17]: we fix dataset A and successively increase the size of dataset B (to a multiple of the size of dataset A).

6.3 Loading the Data

To illustrate that the main performance bottleneck of a spatial join is not the reading of the data into memory, we measure the time it takes to load the data in memory and compare it to performing a spatial join with the PBSM approach. In the experiment we use as dataset A a 1.6M uniform dataset and increase dataset B from 1.6M to 9.6M.

The time to load never exceeds 2 seconds whereas the execution time for the fastest state-of-the-art approach (PBSM-500) grows from 334 to 1512 seconds. The time to load the datasets is hence dwarfed by the time taken for the spatial join. Improving on the time spent on the spatial join therefore will effectively reduce the overall execution time.

6.4 Varying Dataset B

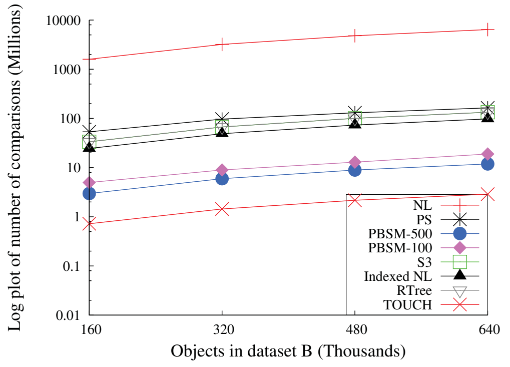
In a first set of experiments we fix ϵ, use a dataset A of fixed size and steadily increase the size of dataset B, similar to the experiments in [17, 27]. To compare TOUCH with all other approaches we first experiment with small datasets B and to study TOUCH in more detail, we experiment with bigger datasets B.

Small Datasets: Figure 8 illustrates the performance of all algorithms for the small uniform datasets. The figure shows the number of object-object comparisons (tests for intersection) and the execution time in logarithmic scale. In this figure dataset A contains $10,000$ objects and the ϵ of the join predicate is 10 while we vary the number of objects of dataset B from $160,000$ to $640,000$ with steps of $160,000$. This experiment shows that our approach and PBSM (both configruations) drastically outperform the nested loop join and plane-sweep in terms of execution time and number of comparisons. Moreover, the results confirm that the execution time directly depends on the number of comparisons.

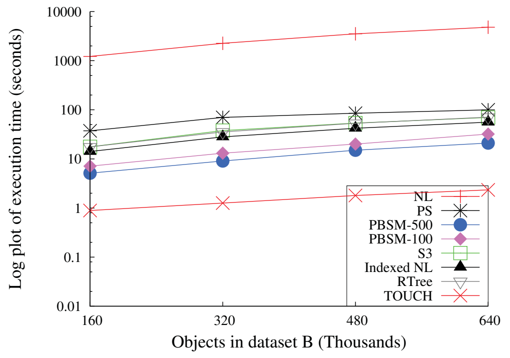
Clearly, execution of the nested loop join is slowest because it requires $O(n^2)$ comparisons. The plane-sweep approach on the other hand has a comparatively high execution time because it has to perform too many comparisons: for example in two dimensions, objects sorted by their x-coordinates may be close to each other on the x-axis but far apart on the y-axis. The plane-sweep hence performs redundant comparisons resulting in a higher execution time.

Because PBSM-100 has fewer and thus bigger grid cells than PBSM-500, it has more objects in each grid cell and hence more objects that are not close to each other will be needlessly compared when two corresponding grid cells are joined. This results in more comparisons and PBSM-100 is therefore slower than PBSM-500.

Large Datasets: in the following set of experiments we fix dataset A to 1.6M and steadily increase the size of dataset B. Figures 9, 10 and 11 illustrate the number of comparisons,



(a) Comparisons



(b) Execution Time

Figure 8: Small uniform datasets, increasing the size of B with $\epsilon = 10$.

execution time and memory consumption of the five remaining algorithms (two indexing approaches, indexed nested loop join (INL) and R-Tree spatial join (RTree), and three non-indexing approaches namely, TOUCH, S3 and PBSM (in two configurations) respectively for uniform, Gaussian and clustered large datasets. Due to the long execution time we exclude the nested loop join and plane-sweep join in these experiments. In terms of running time for all the three types of synthetic datasets, TOUCH is one order of magnitude faster than PBSM-500, PBSM-500 outperforms PBSM-100 and PBSM-500 is also one order of magnitude faster than the three other algorithms. However, PBSM-500 consumes more than two orders of magnitude more memory compared to the other four algorithms. Because the grid cells of PBSM-100 are bigger, fewer elements are replicated and PBSM-100 thus requires less memory.

The performance of the algorithms on different datasets can be explained by comparing the selectivity of the joining datasets. As Table 1 shows, Gaussian datasets have the largest selectivity, followed by clustered datasets and finally the uniform datasets. As a result all the algorithms perform faster on the uniform datasets than on clustered datasets and they perform faster on the clustered datasets than on the Gaussian datasets. Likewise, the algorithms do more comparisons on Gaussian datasets than on the clustered datasets and more on the clustered datasets than on the uniform datasets. The algorithms need approximately the same amount of memory independent of the dataset type.

S3 shows an interesting behavior for the different dataset types compared to the other algorithms. The best performance of S3 can be seen on the uniform datasets, Figure 9, while this algorithm performs worst on the clustered datasets. S3 uses space-oriented equi-width grid partitioning in contrast to TOUCH, which uses data-oriented partitioning; thus, when the objects are not distributed uniformly, as in the clustered datasets, S3 performs worst. This explains why S3 performs faster in comparison to the indexed nested loop join on the uniform datasets, Figure 9, and the Gaussian datasets, Figure 10, while it is slower on the clustered datasets, Figure 11.

By comparing the performance of INL and the RTree regarding the number of comparisons and execution time in Figures 9, 10 and 11 (a & b), respectively, we can conclude that although both require almost the same number of comparisons, the RTree is always faster than INL. The fact that INL requires more time is due to the repeated traversal of the tree as opposed to the synchronous traversal of the RTree.

Comparing the memory consumption of the algorithms, chart (c) in Figures 9, 10 and 11, PBSM-500 consumes about two orders of magnitude ($80\times$) more space in comparison to the other algorithms. This memory consumption of PBSM-500 is mainly because of multiple assignment and hence object replication. PBSM-100, on the other hand, uses fewer and bigger cells and thus replicates fewer objects leading to less memory consumption.

In the experiments for all datasets, TOUCH requires more memory than INL because TOUCH keeps the buckets constructed based on dataset A in addition to the tree while INL only maintains a tree for dataset A. The RTree maintains one tree for each dataset and thus requires more memory than TOUCH as does S3 because it constructs the hierarchical partitioning for each dataset separately.

6.5 Varying Distance Threshold ϵ

In a second set of experiments, two datasets with 1.6M objects are joined with two different ϵ. The execution time measurement for synthetic datasets with different distributions shows that the behavior of the approaches is similar to what we obtained when increasing the size of dataset B. In both cases, increasing ϵ or the size of dataset B, the selectivity is increased.

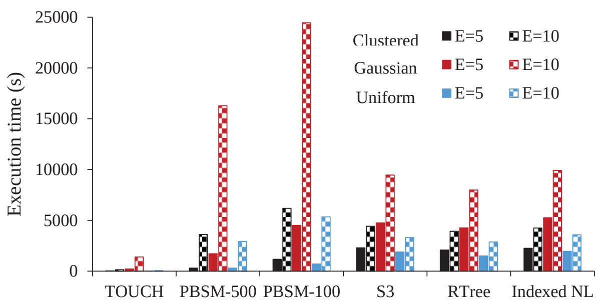
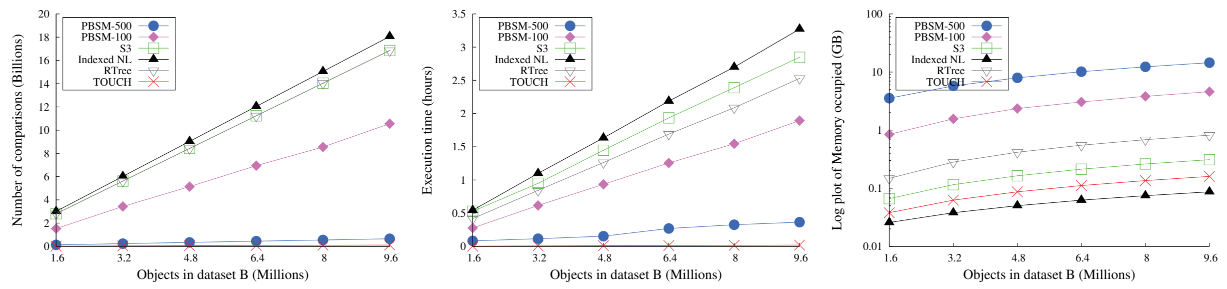


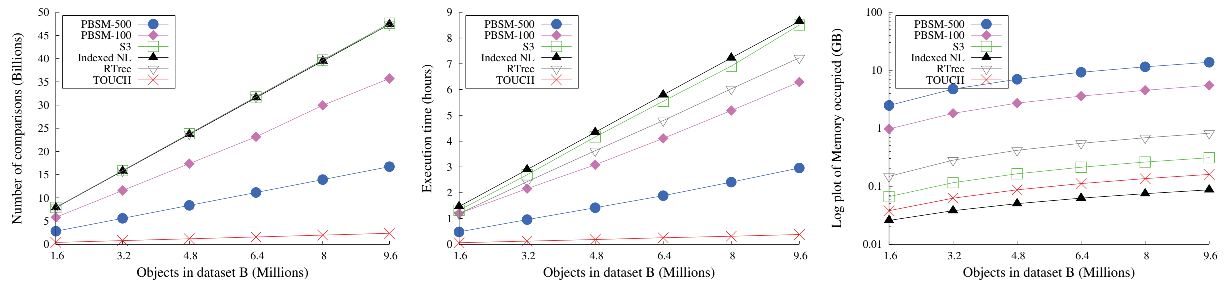
Figure 12: Comparing the approaches for two different ϵ on all datasets.

Figure 12 illustrates the impact of doubling ϵ on the execution time. For most approaches, doubling ϵ leads to an approximate duplication of the execution time. The sole exceptions are both PBSM configurations where an increase of ϵ leads to a disproportionate increase of execution time. This is because, as ϵ grows, PBSM replicates more objects, leading to a super-linear increase in the number of comparisons performed. Because PBSM-100 has bigger cells, an increase of ϵ from 5 to 10 will not lead to as much replication as in the case of PBSM-500, but execution time still grows super-linear due to replication.



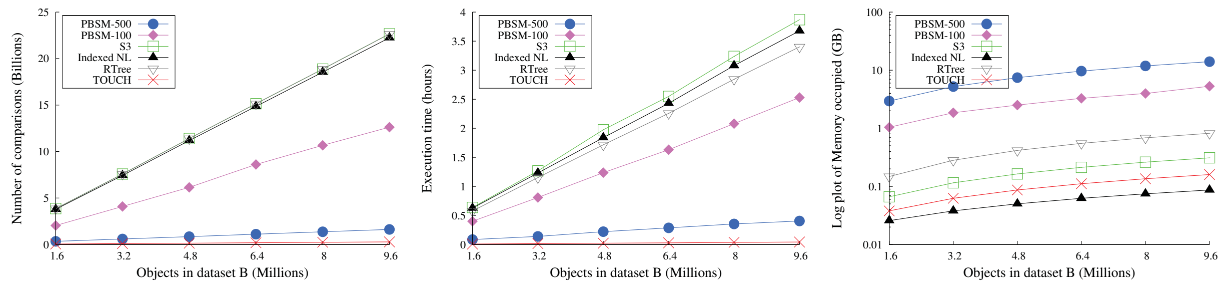
(a) Comparisons (b) Time (c) Memory

Figure 9: Synthetic large uniform datasets varying size of the second dataset with $\epsilon = 5$.



(a) Comparisons (b) Time (c) Memory

Figure 10: Synthetic large Gaussian datasets varying size of the second dataset with $\epsilon = 5$.



(a) Comparisons (b) Time (c) Memory

Figure 11: Synthetic large clustered datasets varying size of the second dataset with $\epsilon = 5$.

6.6 Analysis of TOUCH

In the following set of experiments we analyze the impact of filtering in TOUCH and the fanout parameter.

Filtering: comparing Figures 9 (a) and 11 (a) shows that all algorithms except TOUCH perform more comparisons for the clustered dataset than for the uniform datasets. The reason is the increased selectivity of the clustered dataset compared to the uniform dataset as indicated in Table 1.

However, as can be seen from these experiments (Figures 9 (a) and 11 (a)), TOUCH does not perform many more comparisons for the clustered dataset than for the uniform. As we show in a next experiment this is because of TOUCH's filtering capability. We join dataset A with 1.6M objects with dataset B with an increasing number of objects for the different dataset distributions and a distance predicate ϵ of 5. The results in Figure 13 illustrate that TOUCH filters many objects of dataset B in the clustered dataset (e.g., 440,000 out of the 9.6M objects of dataset B). As explained in Section 4.4, TOUCH can filter the objects of dataset B that do not intersect with any MBR of the inner nodes of the tree and the filtered objects are thus not compared.

Another interesting observation in Figure 13 is the filtering capability of TOUCH on different datasets. For instance when joining two synthetic datasets with 1.6M objects in dataset A and 1.6M objects in dataset B, no object is filtered from the uniform datasets while 0.07 percent of the objects are filtered from the Gaussian datasets and 4.07 percent from the clustered dataset. Hence, the less uniform objects are distributed, the more objects can be eliminated through filtering (and hence the number of comparisons is reduced). The impact of filtering is even more substantial in case of joining neuroscience datasets where 26.58% of the objects can be filtered.

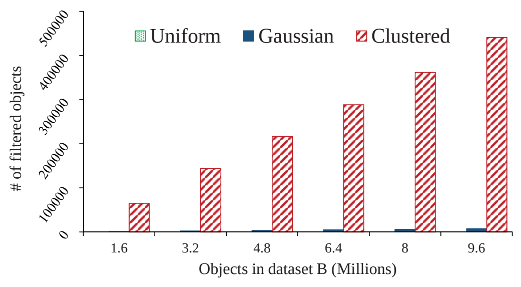
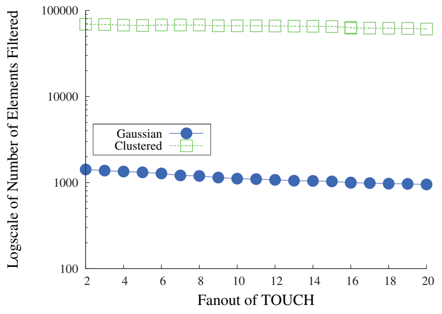


Figure 13: Filtering capability of TOUCH with dataset A of 1.6M with $\epsilon = 5$.

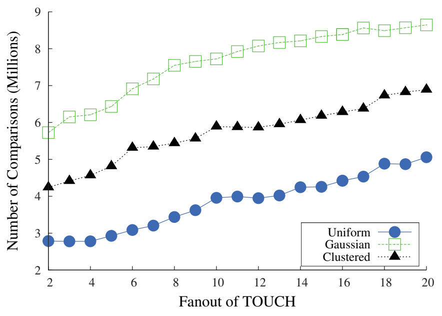
Impact of the Fanout Parameter: in the next experiments we measure the impact of the fanout parameter on

TOUCH by joining dataset A with 1.6M and dataset B with 9.6M (with uniform, normal and clustered distribution; datasets A and B are always of the same type of distribution) and an ϵ of 5. We consecutively increase the fanout from 2 to 20.

In the first experiment shown in Figure 14(a) we measure how many objects can be filtered for an increasing fanout. As mentioned previously, no filtering can be done in case of the uniform distribution. For the other two distributions the trend clearly shows that a smaller fanout allows for more filtering as discussed in Section 5.2. The impact of an increasing fanout, however, does not seem to increase the number of objects filtered significantly (at most less than 1% for the clustered datasets).



(a) Filtering



(b) Comparisons

Figure 14: Impact of the fanout of TOUCH on filtering and number of comparisons with $\epsilon = 5$.

The fanout, however, has a bigger impact on the number of comparisons. As Figure 14(b) shows, the smaller the fanout, the higher the tree and hence the better distributed the elements are, leading to fewer comparisons. The difference is significant: for all distributions, 1.5× fewer comparisons are needed for fanout 2 as opposed to fanout of 20.

6.7 Neuroscience Datasets

Finally, we also measure the performance of TOUCH on neuroscience datasets. True to the neuroscience use case we execute this operation that is executed once per model the neuroscientists build by joining two datasets. Dataset A (the axons) has 644K objects (boxes) and dataset B (the dendrites) with 1.285M objects in a volume of $285\mu m^3$. The average volume of the bound bounding box of the objects in both datasets is $1.34\mu m^3$. The configuration of the approaches is the same as in previous experiments and the best found experimentally (see Section 6.1).

Figure 16 illustrates the same measurements, number of comparisons, execution time and the amount of memory required, on the neuron dataset for two different join predicates (i.e., $\epsilon = 5$ and $\epsilon = 10$). The measurements in this figure show that TOUCH outperforms all other approaches in terms of time and space. Also for these datasets, PBSM-500 is the fastest (after TOUCH) but it requires substantially more memory (rightmost graph in Figure 16), thus limiting its scalability. As with previous experiments, PBSM-100 requires substantially less memory than PBSM-500, but at the same time it requires more comparisons and also more time.

Because the neuroscience datasets are very densely populated in the center, but extremely sparse elsewhere, filtering has a considerable impact. For $\epsilon = 5$, 341,553 objects or 26.58% of the dataset B can be filtered. In case of $\epsilon = 10$, the objects are bigger (because of the distance predicate) and consequently fewer objects can be filtered. Still, a considerable share, 21.23%, is filtered.

In a next experiment we measure how the approaches perform with an increasing density of the spatial model. To emulate increasing density we take the set of axon cylinders and dendrite cylinders from the densest model built to date. In every step we randomly choose an increasing subset of both datasets and join them, emulating increasing density.

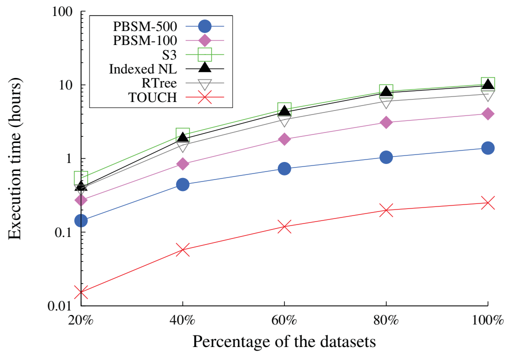


Figure 15: Execution time for increasingly dense spatial neuroscience datasets with $\epsilon = 5$.

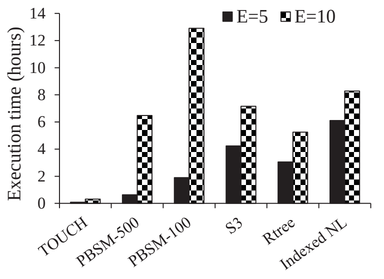
As the results in Figure 15 show, TOUCH already today outperforms the fastest approach among S3 and the R-Tree-based approaches by a factor of 50 for the densest dataset. TOUCH is 8 times faster than the fastest state-of-the-art approach, PBSM-500. At the same time, PBSM-500 requires 6GB of memory while TOUCH only requires 500MB.

7. CONCLUSIONS

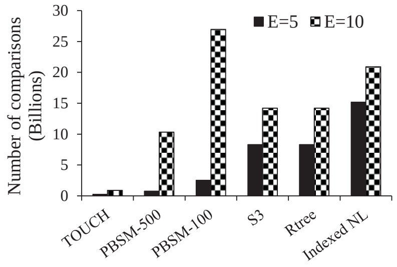
In this paper we identify the lack of efficient approaches for in-memory spatial joins. We demonstrate that the two in-memory approaches, namely the nested-loop and the plane-sweep, are inefficient and that disk-based approaches are not efficient either when applied in main memory. An exception is PBSM, which is fast but needs excessive memory.

We develop TOUCH, a spatial join algorithm that combines the advantages while avoiding the pitfalls of previous approaches. TOUCH avoids a space-oriented partitioning on the large scale and uses a data-oriented partitioning to organize both datasets. To avoid object replication, objects of one dataset are used to construct an R-Tree-like hierarchy, while those of the other are assigned to the level where they are fully contained by an MBR. The partitions on different levels are joined using a space-oriented partitioning.

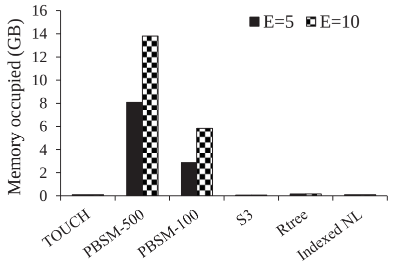
We experimentally demonstrate that our approach is faster and has a substantially smaller memory footprint than the previous state of the art with real and synthetic datasets. Our results also indicate that TOUCH is scalable to larger and denser datasets, a key issue in scaling up the neuroscience application we are motivated from. Thus, TOUCH



(a) Time



(b) Comparisons



(c) Memory

Figure 16: Comparison of all approaches for ϵ of 5 and 10 on neuroscience datasets.

is recommendable as a method of choice for such applications. Furthermore, as we only make a few assumptions about dataset characteristics, with TOUCH we provide a general-purpose solution applicable on any spatial dataset.

8. REFERENCES

[1] W. G. Aref and H. Samet. A Cost Model for Query Optimization Using R-Trees. In *GIS '94*.

[2] W. G. Aref and H. Samet. Cascaded Spatial Join Algorithms with Spatially Sorted Output. In *GIS '96*.

[3] W. G. Aref and H. Samet. Hashing by Proximity to Process Duplicates in Spatial Databases. In *CIKM '94*.

[4] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The Priority R-tree: a practically efficient and worst-case optimal R-tree. In *SIGMOD '04*.

[5] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable Sweeping-Based Spatial Join. In *VLDB '98*.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: an efficient and robust access method for points and rectangles. *SIGMOD Record*, 19(2):322–331, 1990.

[7] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins Using R-Trees. In *SIGMOD '93*.

[8] J.-P. Dittrich and B. Seeger. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *ICDE 2000*.

[9] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 3rd edition, 2000.

[10] A. Farris, A. Sharma, C. Niedermayr, D. Brat, D. Foran, F. Wang, J. Saltz, J. Kong, L. Cooper, T. Oh, T. Kurc, T. Pan, and W. Chen. A Data Model and Database for High-resolution Pathology Analytical Image Informatics. *Journal of Pathology Informatics*, 2(1):32, 2011.

[11] Y. J. García, M. A. López, and S. T. Leutenegger. A Greedy Algorithm for Bulk Loading R-trees. In *GIS '96*.

[12] S. Gnanakaran, H. Nymeyer, J. Portman, K. Y. Sanbonmatsu, and A. E. Garcia. Peptide folding simulations. *Current Opinion in Structural Biology*, 13(2):168–174, 2003.

[13] A. Guttman. R-trees: a Dynamic Index Structure for Spatial Searching. In *SIGMOD '84*.

[14] O. P. Hamill, A. Marty, E. Neher, B. Sakmann, and F. J. Sigworth. Improved Patch-clamp Techniques for High-resolution Current Recording from Cells and Cell-free Membrane Patches. *Pflügers Archiv European Journal of Physiology*, 391:85–100, 1981.

[15] E. H. Jacox and H. Samet. Spatial Join Techniques. *ACM TODS '07*.

[16] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In *VLDB '94*.

[17] N. Koudas and K. C. Sevcik. Size Separation Spatial Join. In *SIGMOD '97*.

[18] J. Kozloski, K. Sfyrakis, S. Hill, F. Schürmann, C. Peck, and H. Markram. Identifying, Tabulating, and Analyzing Contacts Between Branched Neuron Morphologies. *IBM Journal of Research and Development*, 52(1/2):43–55, 2008.

[19] S. Leutenegger, M. Lopez, and J. Edgington. STR: a Simple and Efficient Algorithm for R-Tree Packing. In *ICDE '97*.

[20] M.-L. Lo and C. V. Ravishankar. Spatial Hash-Joins. In *SIGMOD '96*.

[21] M.-L. Lo and C. V. Ravishankar. Spatial Joins Using Seeded Trees. In *SIGMOD '94*.

[22] G. Luo, J. F. Naughton, and C. J. Ellmann. A non-blocking parallel spatial join algorithm. In *ICDE*, pages 697–705, 2002.

[23] N. Mamoulis and D. Papadias. Slot Index Spatial Join. *IEEE TKDE*, 15(1):211–231, 2003.

[24] P. Mishra and M. H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[25] S. Nobari, T.-T. Cao, P. Karras, and S. Bressan. Scalable parallel minimum spanning forest computation. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 205–214, New York, NY, USA, 2012. ACM.

[26] J. Orenstein. A Comparison of Spatial Query Processing Techniques for Native and Parameter Spaces. In *SIGMOD '90*.

[27] J. M. Patel and D. J. DeWitt. Partition Based Spatial-Merge Join. In *SIGMOD '96*.

[28] F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Springer, 1993.

[29] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB '87*.

[30] M. Ubell. The Montage Extensible DataBlade Architecture. In *SIGMOD '94*.